This correspondence is being deposited with the United States Postal Service as Express Mail addressed to: Box Patent Application, Assistant Commissioner for Patents, Washington, D. C. 20231, on May 14, 2001, Express Mail Receipt No. EF055070213US.

NETWORK INTERFACE DEVICE EMPLOYING A DMA COMMAND QUEUE

Stephen E. J. Blightman
Daryl D. Starr
Clive M. Philbrick

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001]This application claims the benefit under 35 U.S.C. §120 of U.S. Patent Application Serial No. 09/464,283, filed December 15, 1999, by Laurence B. Boucher et al., which in turn claims the benefit under 35 U.S.C. §120 of U.S. Patent Application Serial No. 09/439,603, filed November 12, 1999, by Laurence B. Boucher et al., which in turn claims the benefit under 35 U.S.C. § 119(e)(1) of the Provisional Application Serial No. 60/061,809, filed on October 14, 1997. This application also claims the benefit under 35 U.S.C. §120 of U.S. Patent Application Serial No. 09/384,792, filed August 27, 1999, which in turn claims the benefit under 35 U.S.C. § 119(e)(1) of the Provisional Application Serial No. 60/098,296, filed August 27, 1998.

[0002]This application also claims the benefit under 35 U.S.C. §120 of U.S. Patent Application Serial No. 09/067,544, filed April 27, 1998, now U.S. Patent No. 6,226,680. This application also claims the benefit under 35 U.S.C. §120 of U.S. Patent Application Serial No. 09/416,925, filed October 13, 1999. The subject matter of all the above-identified patent applications, and of the two above-identified provisional applications, is

incorporated by reference herein.

BACKGROUND INFORMATION

[0003] Figure 1 (Prior Art) is a diagram of a network interface device (NID) 100 that couples a host computer 101 to a network 102. The particular NID 100 of Figure 1 is an expansion card that is coupled to host computer 101 via PCI bus 103. NID 100 includes an integrated circuit 104, physical layer interface circuitry 105, and dynamic random access memory (DRAM) 106. Integrated circuit 104 includes a media access controller 107, queue manager hardware 108, a processor 109, sequencers 110, an SRAM controller 111, static random access memory (SRAM) 112, a DRAM controller 113, a PCI bus interface 114, a DMA controller 115, a DMA command register 116, and a DMA command complete register 117. For additional detail on one such integrated circuit 104, see U.S. Patent Application Serial No. 09/464,283, filed December 15, 1999 (the subject matter of which is incorporated herein by reference).

[0004] In one example, NID 100 is used to retrieve three portions of data 118-120 from host storage 121 and to output them to network 102 in the form of a data payload of a packet. In the situation where the three portions 118-120 are stored on different pages in host storage 121, it may be necessary to move the data to DRAM 106 individually. The three portions 118-120 are to all be present in DRAM 106 before packet transmission begins.

[0005] In the device of Figure 1, processor 109 causes DMA controller 115 to execute DMA commands 122 by placing the DMA commands into SRAM 112 via lines 123 and SRAM controller 111. Once a DMA command is present in SRAM 112,

processor 109 instructs the DMA controller to fetch and execute the DMA command by setting a bit in DMA command register 116. A DMA command includes a source address for data to be moved, a destination address for data to be moved, and a byte count. In the example of Figure 1, there can be thirty-two pending DMA commands 122 stored in SRAM 112. Consequently, there are thirty-two bits in DMA register 116. DMA controller 115 fetches the command, executes the command, and then sets a corresponding bit in 32-bit DMA command complete register 117. Processor 109 monitors the bit in the DMA command complete register 117 to determine when the DMA command has been completed.

[0006] In the example of Figure 1, each of the three data portions 118-120 is moved into DRAM 106 in two moves. In a first move, the data is moved from host storage 121 and to a buffer in SRAM 112. In a second move, the data is moved from the buffer in SRAM 112 and to DRAM 106.

[0007] To move the first portion of data 118 from host storage 121 to DRAM 106, processor 109 writes a DMA command into SRAM 112. DMA controller 115 retrieves the DMA command from SRAM 112 via SRAM controller 111 and lines 124. DMA controller 115 then executes the command by sending a request to PCI bus interface 114 via lines 125. PCI bus interface 114 responds by retrieving data 118 from a particular address X1 and supplying that data to SRAM controller 111 via lines 126. PCI bus interface 114 reports completion of this action by returning an acknowledge signal to DMA controller 115 via line 127. SRAM controller 111 writes the data into the buffer in SRAM 112. DMA controller 115 then issues a request to DRAM controller 113 via lines 128. Data from SRAM 112 is then transferred via SRAM controller 111, lines 129, and DRAM

controller 113 to a location 130 (address Z1) in DRAM 106. When this action is complete, DRAM controller 113 returns an acknowledge signal to DMA controller 115 via line 131. DMA controller 115 sets a bit corresponding to this completed DMA command in the DMA command complete register 117. In this way, each of the three portions of data 118-120 is moved through SRAM 112 and into DRAM 106. Although the three data portions are illustrated being stored in DRAM 106 at different locations Z1, Z2 and Z3, this need not be the case. All three data portions are sometimes stored in one continuous block.

[0008] In the particular NID 100 of Figure 1, DMA controller 115 may execute DMA commands in an order different from the order in which the DMA commands were placed in SRAM 112 by processor 109. For example, while DMA controller 115 is moving the first portion 118 to DRAM 106, the processor 109 may place additional DMA commands into SRAM 112. When DMA controller 115 finishes moving first portion 118, the DMA controller 115 may fetch a DMA command to move the third portion 120 next. Because all three data portions 118-120 are to be in DRAM 106 before transmission of the packet begins, processor 109 cannot only check that the last move in the sequence is completed. Rather, processor 109 must check to make sure that all the moves are completed before processor 109 goes on in its software to execute the instructions that cause the ultimate packet to be formed and output from NID 100.

SUMMARY

[0009] It may be undesirable for processor 109 on network interface device 100 to have to monitor a set of multiple

DMA commands to make sure that all of the DMA commands in the set have been executed. Monitoring a set of multiple DMA commands can be complex. Monitoring multiple such DMA commands can involve multiple processor instructions and consequently can slow processor execution.

[0010] In accordance with one embodiment of the present invention, a DMA command queue is maintained on a network interface device. A processor on the network interface device pushes values onto the DMA command queue. Although a value pushed onto the queue can be a DMA command itself in some embodiments, each value pushed preferably includes an address that identifies a location where a particular DMA command is stored. The DMA command queue is popped such that a DMA controller on the network interface device executes DMA commands in the order in which the processor pushed the associated values onto the DMA command queue. Complications associated with DMA commands being issued by the processor in one order but being completed by the DMA controller in another order are thereby alleviated or avoided.

[0011] In one embodiment, a DMA command complete queue is also maintained by dedicated queue manager hardware on the network interface device. Upon completing a DMA command, the DMA controller pushes a value onto the DMA command complete queue. The value identifies the DMA command completed. The processor determines which DMA commands have been completed by popping the DMA command complete queue. In a situation where a set of multiple DMA commands must be executed before software executing on the processor takes a particular software branch, the processor does not monitor each of the multiple DMA commands to make sure that all of the commands have been completed. Rather, the

processor monitors the values popped off the DMA command complete queue. When the value popped off the DMA command complete queue is the value indicative of the last DMA command of the multiple DMA commands, then the processor takes the particular branch. In one embodiment, the multiple DMA commands are commands to move portions of data from different pages in host storage onto the network interface device, where those portions of data are to be put together on the network interface device to form a data payload of a network packet. All the portions of data are moved onto the network interface device before the processor takes the software branch that causes the packet to be formed and output from the network interface device.

[0012] In some embodiments, the network interface device maintains multiple DMA command queues. Some of the DMA command queues are for moving data from one location on the network interface device to another location on the network interface device. For example, multiple frames of a session layer message are, in one embodiment, received onto the network interface device and stored in DRAM. A first part of each of the frames is then moved from DRAM to SRAM (from a first location on the network interface device to a second location on the network interface device, the first location being the DRAM and second location being the SRAM), where each DMA move is carried out by a DMA controller on the network interface device executing an associated DMA command from a DMA command queue. The DMA command queue is maintained by hardware queue manager circuitry on the network interface device. Alternatively, the DMA command queue is a software queue maintained on the network interface device. The first parts of the frames are moved into SRAM so that a processor on the network

interface device can then examine the headers of the frames and determine from those headers what to do with the associated frames. In some embodiments, there is a separate DMA controller for each DMA command queue.

[0013]Other embodiments are described. This summary does not purport to define the invention. The claims, and not this summary, define the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0014]Figure 1 (Prior Art) is a simplified diagram of one particular prior art network interface device.

[0015]Figure 2 is a diagram of a network interface device (NID) in accordance with one embodiment of the present invention.

[0016]Figure 3 is a flowchart of a method in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0017]Figure 2 is a diagram of a network interface device 200 in accordance with one embodiment of the present invention. Network interface device 200, in this example, is an intelligent network interface card (INIC) that is used to couple host computer 201 to a network 202. The network interface device (NID) 200 is coupled to host computer 201 via bus 203. Bus 203 in this example is a PCI bus.

[0018]Although not illustrated in Figure 2, bus 203 itself may not extend all the way to host storage 204. Rather, an intermediate integrated circuit (for example, a memory

controller integrated circuit, or an I/O integrated circuit, or a bridge integrated circuit such as the Intel 82801) may be disposed between the bus 203 and host storage 204. In such a case, host storage 204 may be coupled to the intermediate integrated circuit via the local bus of the host computer 201.

[0019]The particular network interface device 200 of Figure 2 includes physical layer interface circuitry 205, an integrated circuit 206, and local memory 207. In this example, local memory 207 is dynamic random access memory (DRAM). Integrated circuit 206 includes media access control circuitry 208, queue manager hardware 209, a processor 210, a sequencers block 211, an SRAM controller 212, static random access memory 213, a DRAM controller 214, a PCI bus interface 215, and a DMA controller 216. The diagram of Figure 2 is a simplified diagram in order to facilitate explanation of an operation of network interface device 200. The network interface device 200 contains other circuit blocks and other interconnections.

[0020]Network interface device 200 performs transport and network layer protocol processing on certain network traffic using hardware specially-designed to speed such protocol processing (fast-path processing), thereby accelerating the processing of this traffic and thereby off-loading the main protocol processing stack (slow-path processing) on the host computer. For additional details on the structure and operation of one particular embodiment of the network interface device 200, see U.S. Patent Application Serial No. 09/464,283, filed December 15, 1999 (the subject matter of which is incorporated herein by reference). For additional details on the structure and operation of one embodiment of queue manager hardware 209,

see U.S. Patent Application Serial No. 09/416,925, filed October 13, 1999 (the subject matter of which is incorporated herein by reference).

[0021] Queue manager hardware 209 maintains a DMA command queue 217 and a DMA command complete queue 218. Although the DMA command queue 217 and the DMA command complete queue 218 are illustrated in Figure 2 as being disposed within queue manager 209, the two queues 217 and 218 are in this embodiment actually stored in SRAM 213 and DRAM 207. The head and tail of each queue are stored in SRAM 213, whereas the body of each queue is stored in DRAM 207. Storing the heads and tails in fast SRAM rather than relatively slow DRAM speeds pushing and popping of the queues. It is to be understood, however, that the queues 217 and 218 maintained by queue manager hardware 209 can be stored in any of numerous possible places, including but not limited to the queue manager 209, SRAM 213, and DRAM 207.

[0022] Processor 210 can cause values to be pushed onto the DMA command queue 217 and can cause values to be popped off the DMA command complete queue 218. DMA controller 216 can cause values to be popped off the DMA command queue 217 and can cause values to be pushed onto the DMA command complete queue 218.

[0023] In one example, processor 210 proceeds down a software thread whereby a first portion of data 219 (at address X1) and a second portion of data 220 (at address X2) respectively in host storage 204 are to be moved into local memory 207 to locations Z1 and Z2, respectively. Only after both of the data portions 219-220 are present in local memory 207 does the software executed by processor

210 branch into the instructions that cause the two data portions to be output from the network interface device 200. After the branch, the two data portions 219 and 220 are output in the form of a data payload (or a portion of a data payload) of a network communication such as a TCP/IP packet. The network communication is output via media access control circuitry 208 and physical layer interface circuitry 205.

[0024] In the illustrated embodiment, there can be up to thirty-two pending DMA commands 221 stored in SRAM 213. Although the DMA commands 221 are illustrated in Figure 2 as being located in one contiguous block, the various DMA commands 221 may be located in various different parts of SRAM 213. Each DMA command in this example includes 128 bits. The 128 bits include a source address (where to get data to be moved), a destination address (where to place the data), and a byte count (number of bytes of data to be moved).

[0025] Figure 3 is a flow chart that illustrates a method in accordance with an embodiment of the invention. Processor 210, as it proceeds down the thread, places a set of multiple (in this case two) DMA commands into SRAM 213. The first DMA command is to move first portion of data 219 from address X1 in host storage 204 to address Z1 in local memory 207. After placing the first DMA command into SRAM 213 via lines 222 and SRAM controller 212, processor 210 causes a value that points to the first DMA command in SRAM 213 to be pushed (step 300) onto DMA command queue 217. Processor 210 does this by first writing a queue identifier into a queue control register (not shown) in queue manager 209. This queue identifier indicates the queue onto which a subsequently written 32-bit queue data value is to be

pushed. Processor then writes a 32-bit queue data value onto a queue data register (not shown) in queue manager 209. The 32-bit queue data value contains: a 16-bit address where the DMA command is located in SRAM 213, a 5-bit termination queue identifier, an 8-bit termination value to be pushed onto the queue identified by the termination queue identifier upon completion of the DMA command, a one-bit DMA chain bit value, a one-bit dummy DMA value, and one unused bit. If the DMA chain bit is set, this indicates the when the DMA controller has completed the DMA command, that the DMA controller is not to push a termination value onto any queue. If the dummy DMA bit is set, this indicates that the DMA controller is not to perform a DMA command per se but is nevertheless to push the 8-bit termination value onto the queue identified by the 5-bit termination queue identifier.

[0026]By placing DMA commands into SRAM 213, writing to the queue control register and then writing to the queue data register, processor 209 pushes two values onto DMA command queue 217. The first value is indicative of a first DMA command to move the first portion of data 219 from host storage 204 to local memory 207. The second value is indicative of a second DMA command to move the second portion of data 220 from host storage 204 to local memory 207.

[0027]DMA controller 216 causes a value to be popped off (step 301) the DMA command queue 217. In the present example, this first value points to the location in SRAM 213 where the first DMA command is stored. DMA controller 216 retrieves the first DMA command via SRAM controller 212 and lines 223 and executes the first DMA command. DMA controller 216 issues a request to PCI bus interface 215

via lines 224. PCI bus interface 215 in turn retrieves the first portion of data 219, supplies that first portion of data to SRAM controller 212 via lines 225, and returns an acknowledge signal to DMA controller 216 via line 226. SRAM controller 212 in turn writes the first portion of data 219 into SRAM 213. DMA controller 216 then issues a request to DRAM controller 214 via lines 227 to place the first portion of data into DRAM 207. The first portion of data passes from SRAM 213 via SRAM controller 212, lines 228, and DRAM controller 214 to DRAM 207. When the move is complete, DRAM controller 214 returns an acknowledge signal to DMA controller 216 via line 229. When the first data is present in local memory 207 at address Z1, the first DMA command is complete. DMA controller 216 pushes the termination value for the first DMA command onto the DMA command complete queue 218 (DMA command complete queue 218 is the queue identified by the termination queue identifier associated with the first DMA command).

[0028] The pushing of values onto the DMA command queue, the popping the values off the DMA command queue, and the executing of DMA commands indicated by the popped values is repeated (step 302) in such an order and fashion that the first and second portions of data are transferred to local memory. In the presently described example, two values are pushed onto the DMA command queue, then the first value is popped from the DMA command queue, and then the second value is popped from the DMA command queue. This need not be the case, however. The first value may be pushed onto the DMA command queue, then the first value may be popped from the DMA queue, then the second value may be pushed onto the DMA command queue, then the second value may be popped from the DMA command queue. Both orders of pushing

and popping are encompassed by the flowchart of Figure 3. Step 300 can be carried out twice before step 301 is carried out. Alternatively, step 300 can be carried out once, then step 301 can be carried out once, then step 300 can be carried out a second time, and then step 301 can be carried out a second time.

[0029] In the presently described example, DMA controller 216 retrieves the second DMA command and executes the second DMA command such that the second portion of data 220 is moved into local memory 207 via SRAM 213. When the second portion of data is present in local memory 207, DMA controller 216 pushes the termination value for the second DMA command onto the DMA command complete queue 218 (DMA command complete queue 218 is the queue identified by the termination queue identifier associated with the first DMA command).

[0030] Only after both portions of data 219 and 220 are present on the network interface device 200, does processor 210 branch to the instructions that cause the data of portions 219 and 220 to be output from the network interface device in the form of a data paylaod of a network communcation. To ensure that all of the necessary data portions have been moved to the network interface device 200, processor 210 need not monitor the complete status of all the DMA commands involved as in the prior art example of Figure 1. Rather, processor 210 polls a 32-bit queue-out-ready register (not shown) in queue manager 209. Each bit in the queue-out-ready register indicates whether there is a value present in a corresponding queue. Processor 210 therefore polls the queue-out-ready register, determines that the bit for the DMA command complete queue is set, and then pops the DMA command complete queue 218. Processor

210 repeats this process until the value popped off the DMA command complete queue 218 is the value indicative of the last DMA command in the set of DMA commands. When the value popped off the DMA command complete queue is the value indicative of the last DMA command in the set of DMA commands, then it is known that all the proceeding DMA commands in the set have been executed.

[0031] In the presently described example, processor 210 monitors the values popped from DMA command complete queue 218 and when the value popped is the termination value for the second DMA command, then processor 210 (step 303) executes instructions that cause the first and second data portions 219 and 220 to be output from the network interface device 200. The data from local memory 207 passes via lines 230 to a transmit sequencer in sequencers block 211, and then through media access control circuitry 208 and physcial layer interface circuitry 205 to the network 202.

[0032] Where a set of multiple DMA commands are to be executed before processor 210 takes a software branch, processor 210 can be pushing values onto the DMA command queue 217 at the same time that DMA controller 216 is executing DMA commands. Although the movement of data from host storage 204 to local memory 207 involves temporary storage in SRAM 213 in this example, there need not be any such temporary storage. For example, in some embodiments data is transferred directly from host storage 204 to local memory 207. Data being transferred is buffered in SRAM 213 in the embodiment of Figure 2 because local memory 207 is DRAM that may not be immediately available to store data when the data is transferred onto the network interface device. Rather than having to wait for the DRAM to become

available before writing the data onto the network interface device, the data is written into SRAM 213. The data is later moved to DRAM when the DRAM is available.

[0033]Although the functionality of the network interface device 200 is described here as being carried out on an expansion card, it is to be understood that in some embodiments the network interface device is part of the host computer itself. In some embodiments, the network interface device is realized on a motherboard of the host computer. In some embodiments, the network interface device is integrated into a memory controller integrated circuit of the host computer such that data from host storage 204 is transferred to the network interface device without passing over a PCI bus or to an expansion card. For example, the network interface device may be integrated into the Intel 82815 Graphics and Memory Controller Hub, the Intel 440BX chipset, or the Apollo VT8501 MVP4 North Bridge chip. In some embodiments, the network interface device is part of an I/O integrated circuit chip such as, for example, the Intel 82801 integrated circuit of the Intel 820 chip set. In some embodiments where the network interface device functionality is embodied in an I/O integrated circuit chip, data from host storage 204 is transferred to the network interface device without passing over a PCI bus or to an expansion card. Rather the data passes through the host computer local bus, onto the I/O integrated circuit chip, and from the I/O integrated circuit chip's network interface port substantially directly to network 202 (through a physical layer interface device (PHY)) without passing over any expansion card bus.

[0034]Although the present invention has been described in connection with certain specific embodiments for

instructional purposes, the present invention is not limited thereto. The use of a DMA command queue on a network interface device to alleviate or solve problems associated with DMA commands being completed out of order (in an order other than the order in which the processor issued the DMA commands) is not limited to the transfer of data from host storage 204 to network interface device 200. A DMA command queue is employed in some embodiments to move data from one location on network interface device 200 to another location on network interface device 200. Queue manager hardware 209 may maintain multiple DMA command queues at the same time. A separate DMA controller may be provided on the network interface device 200 for each such DMA command queue. Accordingly, various modifications, adaptations, and combinations of various features of the described embodiments can be practiced without departing from the scope of the invention as set forth in the claims.